CSCE 2014 – Programming Project 1

Midpoint Due Date – 6/2/17 at 11:59 PM
Final Due Date – 6/8/17 at 11:59 PM

1. Problem Statement:

For this assignment, your task is to read into memory a description of a maze from a text file. Your program will then attempt to find a path through the maze using a recursive backtracking algorithm. If your program is successful, it will print the solution to the maze (that is, the path that should be taken). If your program is unsuccessful, a message will be printed to the console informing the user that no such solution to the maze exists.

Example Input/Output

Assuming `maze.txt` looks like this:

```
* _ * * * * * * * *
* _ _ _ _ _ * _ _ *
* _ * * * _ _ _ * *
* _ * _ _ _ * _ * *
* * * * * _ * _ _ *
* _ _ _ _ _ * * * *
* * * _ * * * * _ *
* _ * _ * * _ _ _ *
* _ _ _ _ _ _ * _ *
* * * * * * _ * * *
```

This is what your program should output:

```
Please enter the name of the maze file:
maze.txt
Maze:
* _ * * * * * * * *
* _ _ _ _ _ * _ _ *
* _ * * * _ _ _ * *
* _ * _ _ _ * _ * *
* * * * * _ * _ _ *
* _ _ _ _ _ * * * *
* * * _ * * * * _ *
* _ * _ * * _ _ _ *
* _ _ _ _ _ _ * _ *
* * * * * * _ * * *
```

```
Solution:
* ! * * * * * * * *
* ! ! ! ! ! * _ _ *
* _ * * * ! _ _ * *
* _ * _ _ ! * _ * *
* * * * * ! * _ _ *
* _ _ ! ! ! * * * *
* * * ! * * * * _ *
* _ * ! * * _ _ _ *
* _ _ ! ! ! ! * _ *
* * * * * * ! * * *
```

If the maze actually has no solution, like in this example:

```
* _ * * * * * * * *
* _ _ _ * _ * _ _ *
* _ * * * _ _ _ * *
* _ * _ _ _ * _ * *
* * * * * _ * _ _ *
* _ _ _ _ _ * * * *
* * * _ * * * * _ *
* _ * _ * * _ _ _ *
* _ _ _ _ _ _ * _ *
* * * * * * _ * * *
```

The program should report that there is no solution to the problem:

```
Please enter the name of the maze file:
maze.txt
Maze:
* _ * * * * * * * *
* _ _ _ * _ * _ _ *
* _ * * * _ _ _ * *
* _ * _ _ _ * _ * *
* * * * * _ * _ _ *
* _ _ _ _ _ * * * *
* * * _ * * * * _ *
* _ * _ * * _ _ _ *
* _ _ _ _ _ _ * _ *
* * * * * * _ * * *

No solution.
```

Simplifying Assumptions

We will make several simplifying assumptions to make this problem a bit easier to manage:

- All mazes are of size 10 cells x 10 cells.
- The characters for walls and empty space are fixed. Walls will be represented using '*' characters, and empty space will be represented using '_' characters.
- The solution path will be denoted with a '!' character.
- If there are multiple solutions to a particular maze, your program need only find one.

Rules

The following rules must be observed by your program:

- In finding a solution, you are **only** allowed to move from one free space cell to another neighboring free space cell. At no point are you allowed to step onto or over a wall character.
- Your path may **NOT** cross diagonals. You may only move North, East, South, or West.
- The final path will **NOT** have any branches.
- A well-formed maze should have exactly two entry points along the outer edges, but you should **NOT** assume that every maze will be well-formed. In particular, it will be possible for mazes to have fewer than two entry points (meaning there is no solution to the maze) or more than two entry points (meaning there is likely more than one solution to that particular problem).
- You **cannot** change the original maze (no cheating).
- Your solution **MUST** explicitly use recursion to solve the problem.

2. Design:

For this homework assignment, you will be provided with a skeleton C++ source file (`maze.cpp`) that contains all of the function prototypes you will need, as well as a complete main program. Your job will be to fill in the bodies of each of the auxiliary functions. **You are not to modify the prototypes of any of the functions that have been provided, but you are allowed to create new functions if you believe they should be included**. The functions you must implement are listed below:

1. `bool loadMaze(const string& filename, char maze[MAX_ROWS][MAX_COLS])`

The purpose of this function is to open the file whose name was provided by the user (`filename`) and read the contents into the 2D array named `maze`. This function is expected to address exceptional circumstances using the return value. It will return true when the file read succeeds and false if it fails.

2. `void printMaze(const char maze[MAX_ROWS][MAX_COLS])`

Prints the given maze to the console. You may make your output "pretty", but it is not required.

3.
```
bool validMove(const int r, const int c,
    const char maze[MAX_ROWS][MAX_COLS],
    const bool seen[MAX_ROWS][MAX_COLS])
```

This function should be called by `solveHelper()`. This function will let us know if a desired move (r, c) is feasible. We are not allowed to step outside the bounds of the maze, step onto a wall, or step to a location we've already visited.

4.
```
bool entryPoint(const int r, const int c,
    const char maze[MAX_ROWS][MAX_COLS])
```

This function will be called by `findEntryPoint()` and `solveHelper()`. This function will return true if the given cell (r, c) is an entry point. An entry point is a cell on the outer edge of the map that's not a `WALL` character.

5. `bool findEntryPoint(const char maze[MAX_ROWS][MAX_COLS], int& startRow, int& startCol)`

Searches the maze for a single entry point (by calling the `entryPoint()` function). If one is found, we will save the corresponding row and column in `startRow` and `startCol`. This function allows our solver to know where to start the recursion.

6. `bool solve(const char maze[MAX_ROWS][MAX_COLS], char solution[MAX_ROWS][MAX_COLS])`

This function starts off the recursion, but it is not recursive itself. Like many other recursive functions, it uses a helper function to do the real work. `solve()`'s purpose is to figure out where the recursion should start and pass the appropriate parameters to the helper so the helper can do the hard work. The solution, if there is one, should be placed in `solution`.

7.
```
bool solveHelper(
    const char maze[MAX_ROWS][MAX_COLS],
    const int startRow, const int startCol,
    const int r, const int c,
    bool seen[MAX_ROWS][MAX_COLS],
    char solution[MAX_ROWS][MAX_COLS])
```

This function implements the recursive backtracker. The actual algorithm is outlined below. It takes as arguments the `maze` to search, the location of the initial entry point (`startRow, startCol`), a current cell (`r`, `c`), an array telling us which cells have been seen so far (`seen`), and the `solution`, which will be filled in with the final path when the function finishes. solveHelper() should return true if a path was found and false if all options were exhausted.

Recursive Backtracking Algorithm

The general strategy for any recursive algorithm is to attempt to reduce the problem to a simpler one. When solving a maze, assume we have some notion of state. We know where we are within the maze at all times. We then have two options: either we have reached one of the maze's exit points, or we have not. If we have reached an exit point, we know we have found a path through the maze, and so we were successful in our attempt to find a solution to the problem. If we have not reached an exit point, we just have to take a step towards the goal and reevaluate our stopping criteria. Each time we take a step, we save our location so we'll be able to remember the exact path we took to get to this point.

While this general strategy is sufficient for very simple paths, it doesn't tell us what to do if we encounter a branch (a point where there are multiple directions we could potentially move). This is where a strategy called "backtracking" becomes important. Backtracking means we will save our location every time we have a decision to make (e.g. should I move right or down?). We will choose one of the possible actions and proceed along that path. If at any point we realize that this was not the correct option, we will go back to the branch point and take one of the other available paths. Given enough time, we will eventually reach every cell in the maze that is reachable, including the maze's exit, if one exists.

If we combine these two ideas, recursive traversal and backtracking, we are left with an efficient mechanism for finding paths through arbitrary graphs (like our simple maze). The following pseudocode illustrates the idea:

```
solveHelper(maze, row, col, seen, solution)
{
  // Base case – We've found the way out!
  If (row, col) is an entry point (but not the one we started at)
    Add (row, col) to the solution
    Return true

  // Recursive case – We have to keep searching
  Else
    Mark (row, col) as seen

    For each cardinal direction, dir, in {N, E, S, W}
      newRow = apply(dir, row)
      newCol = apply(dir, col)

      // Call solveHelper() recursively to determine if there's a path from
      // (newRow, newCol) to the exit. If there is, we know this cell (row, col)
      // is part of the solution.
      If (newRow, newCol) is valid AND
        solveHelper(maze, newRow, newCol, seen, solution)
        Add (row, col) to the solution
        Return true

    // We have nowhere left to go. We must have hit a dead end. Time to backtrack!
    Return false
}
```

Extra Credit Opportunities

As written so far, this design is fairly restrictive. Mazes are of a fixed size, they are square, only certain characters may be used in the description file, and the solver only finds at most one solution to the given problem. Students who desire more challenge may attempt to increase the robustness of their solutions by eliminating one or more of the simplifying assumptions listed above in exchange for **a maximum of 5 points of extra credit.**

In particular, students may modify their code to work with **mazes of arbitrary dimensions**, rather than simply 10x10, either by statically allocating a larger block (using only some of the cells), or by dynamically allocating the grid. In this case, you will have to pass the maze dimensions to each relevant function and be more careful about checking for boundary conditions.

Students may also **allow differing characters to be read from the file**. One means of handling this is to modify the file format so that the first line contains the characters that will be used for the walls and for empty spaces. You would have to pass those characters to each function that needs them, however. You will no longer be allowed to use the global constant approach.

Finally, students may modify the backtracking algorithm to **save ALL viable paths through the maze** instead of simply one. This is the hardest of the extra credit opportunities, as it would require modifying the backtracking algorithm and maintaining many solutions (e.g. an array) or otherwise disambiguating each of the possible solutions to the problem. Students wishing to attempt this would have to modify the conditions for stopping the recursive process. You would not stop when you find a solution. You instead

would add the new solution to the collection of solutions and continue until all possible paths have been explored. When the process completes, you will have explored every possible path through the maze completely. At that point, you would print all solutions to the problem to the console, rather than simply one.

3. Implementation:

At the top of `maze.cpp`, we have declared several constants. `MAX_ROWS` and `MAX_COLS` should be used to represent the size of the maze. `WALL`, `SPACE`, and `PATH` are the characters that represent valid states of each cell in the maze. You are expected to use these constants where applicable in your implementation of each of the functions.

We will represent the maze as a 2D array of characters of size `MAX_ROWS` by `MAX_COLS`. Each cell will initially be either `WALL` or `SPACE`. After we load the maze into memory, that array will never be modified. Instead, we will write our solution to another 2D array named `solution`. The `PATH` character will be used to notate the path your solution takes through the maze ('!').

It is suggested that you complete each function one at a time. That will allow you to know immediately where to look if you notice a problem. Also make sure to test each function in isolation as you're writing. Do NOT attempt to implement all functions at once and then debug your code. The process will take considerably longer, and you're more likely to make mistakes. In addition, do everything you can to make your code "bullet-proof". Try providing inputs that are egregiously wrong to see what your code does, for example. Stress tests like these are helpful for finding weaknesses and vulnerabilities. By eliminating them from the start, your code will be more robust and reliable.

4. Style

Make sure your code adheres to the guidelines provided in the Style Guide (available on Moodle). Your goal is to create code that is concise, descriptive, and easy for other humans to read. Avoid typos, spelling mistakes, or anything else that degrades the aesthetic of your code. Your final submission should be work that you are proud to call your own.

5. Testing:

Test your program to check that it operates correctly for all of the requirements listed above. Also check for the error handling capabilities of the code. Try your program with several input values, and save your testing output in text files for inclusion in your project report.

6. Documentation:

When you have completed your C++ program, write a short report using the project report template describing what the objectives were, what you did, and the status of the program. Does it work properly for all test cases? Are there any known problems? Save this report to be submitted electronically.

7. Project Submission:

In this class, we will be using electronic project submission to make sure that all students hand their programming projects and labs on time, and to perform automatic plagiarism analysis of all programs that are submitted.

When you have completed the tasks above go to Moodle to upload your documentation (a single **.pdf** file), and all C++ program files (**.h** and **.cpp**). Make sure your proof of testing is included in the documentation or is submitted as a separate file. Do NOT upload an executable version of your program.

The dates on your electronic submission will be used to verify that you met the due date above. Late projects will receive **NO** credit. You will receive partial credit for all programs that compile even if they do not meet all program requirements, so make sure to submit something before the due date, even if the project is incomplete.

8. Academic Honesty Statement:

Students are expected to submit their own work on all programming projects, unless group projects have been explicitly assigned. Students are NOT allowed to distribute code to each other, or copy code from another individual or website. Students ARE allowed to use any materials on the class website, or in the textbook, or ask the instructor for assistance.

This course will be using highly effective program comparison software to calculate the similarity of all programs to each other, and to homework assignments from previous semesters. Please do not be tempted to plagiarize from another student.

Violations of the policies above will be reported to the Provost's office and may result in a **ZERO** on the programming project, an **F** in the class, or suspension from the university, depending on the severity of the violation and any history of prior violations.